Implémentation d'un modèle de contrôle d'accès pour les documents RDF.

Léo Letouzey (leo.letouzey@upf.pf)*
Alban Gabillon (alban.gabillon@upf.pf)*

Résumé : Dans cet article nous présentons l'implémentation du nouveau modèle de contrôle d'accès pour les documents RDF que nous avons récemment développé. Cette implémentation se présente sous la forme d'un serveur proxy qui permet aux utilisateurs de partager et de protéger leurs documents RDF. Notre serveur propose à la fois une interface de requête pour interroger les documents RDF et une interface d'administration permettant de définir les politiques de sécurité sur ces documents.

Mots Clés : RDF, Modèle de contrôle d'accès, Implémentation

1 Introduction

L'adoption rapide du format RDF[AB04] pour décrire les données et les liens entre elles est à la base de l'émergence du Web Sémantique. Le Web Sémantique repose essentiellement sur cette notion de *sens* qui peut être parfaitement représentée à l'aide de documents RDF.

Avec l'adoption grandissante du format RDF pour représenter des informations dans le web, plusieurs langages d'interrogation de ces documents sont apparus (SERQL[BK03], RDQL[Sea04], RQL[KAC+02],SPARQL). Depuis janvier 2008, SPARQL[PS08] est devenu une recommandation du w3c pour effectuer des requêtes sur des documents RDF. SPARQL est un accronyme récursif pour *SPARQL Protocol And RDF Query Language*. SPARQL peut être utilisé pour effectuer des requêtes sur les documents RDF (les informations peuvent être stockées nativement au format RDF ou être vues comme des données RDF au travers d'une application *middleware*). À l'aide de SPARQL il est possible de rechercher des motifs de graphes obligatoires et/ou optionnels, leurs conjonctions et leur disjonctions.

Une requête SPARQL peut être de quatre types différents, SELECT, CONSTRUCT, ASK ou DESCRIBE. Le premier type ne retourne que les valeurs des variables demandées (et leurs liens éventuels), le second retourne *vrai* ou *faux* en fonction de la présence ou non du motif de graphe demandé dans le document RDF. Les deux derniers retournent des graphes RDF[BB08]. Une requête de type CONSTRUCT retourne un graphe utilisant le motif présent dans la requête et les informations présentes dans le document RDF. Une requête de type DESCRIBE retourne quant à elle un graphe contenant l'ensemble des triplets se rapportant à certains élements choisis dans le document RDF.

* Laboratoire Gepasud, Université de la Polynésie Française - BP 6570, 98702 FAA'A, Polynésie Française
Tel : (+689) 866 438, (+689) 803 880

De nombreux travaux se sont attachés à sécuriser les documents RDF. Mais on retrouve systématiquement deux limites dans ces modèles. Premièrement ils n'utilisent pas d'outils permettant une définition concise des règles de sécurité. En effet, ces règles de contrôle d'accès sont définies à l'aide de motifs de triplet RDF[Red05] [ALC+07] [ACH+09] [Jai06] [DA06] définissant à chaque fois un seul type de triplet à protéger. L'utilisation de ces motifs de triplet permet de disposer d'une politique de sécurité très précise, mais cela devient rapidement difficile à gérer dans le cas d'un document RDF contenant un grand nombre de triplets. Le nombre de règles à exprimer peut rapidement devenir excessif. Deuxièmement, aucun modèle ne dispose d'un système d'administration décentralisé permettant la mise à jour des différentes règles de sécurité. Nous avons récemment présenté un nouveau modèle de sécurité pour contrôler l'accès aux documents RDF[GL10]. Ce modèle est destiné à contrôler les actions SPARQL (SELECT, CONSTRUCT, ASK et DESCRIBE).

Le but de ce papier est de présenter une implémentation de notre modèle de contrôle d'accès. Nous avons développé un serveur proxy qui permet aux utilisateurs de distribuer leurs documents RDF et de gérer l'accès aux données[1].

La suite de l'article est organisée comme suit. La deuxième section présente différents prototypes existants protégeant les données RDF sur le web. Dans la troisième section nous présentons les principales caratéristiques de notre modèle de contrôle d'accès. Dans la section 4, nous présentons les détails de notre prototype de serveur de données RDF sécurisé. Enfin dans la dernière section nous concluons cet article et présentons quelques pistes à étudier pour l'amélioration du modèle de contrôle d'accès et de notre prototype.

2 Travaux existants et motivations

2.1 Travaux existants

Rappelons tout d'abord que les formats RDF et XML, bien que très semblables en apparence sont suffisamment éloignés pour que les solutions élaborées pour le format XML ne soient pas applicables au format RDF. En effet les documents XML ne contiennent pas les informations sémantiques d'un document RDF. C'est cette sémantique supplémentaire qui rend inapplicables les solutions proposées pour les documents XML. De nombreux auteurs s'accordent sur le fait que les modèles de contrôle d'accès sur XML ne peuvent s'adapter à RDF[Red05][Jai06].

Les différents travaux relatifs aux modèles de contrôle d'accès étant déjà discutés dans [GL10], nous ne rappelons que brièvement les conclusions.

Nous avons vu dans [GL10] que les modèles de contrôle d'accès aux documents RDF présentaient deux inconvénients majeurs. Tout d'abord le manque de concision et d'expressivité dans la définition de la politique de sécurité. Ensuite, l'absence de système d'administration de la politique de sécurité. Nous présentons maintenant plusieurs implémentations de modèles de sécurité pour les documents RDF.

Dans [ALC+07], les auteurs présentent une implémentation de leur modèle de contrôle d'accès. Leur implémentation repose sur la ré-écriture des requêtes. Elle se décompose en trois modules distincts, le *Query Extension*, le *Policy Engine* et le *RDF Repository*. Le module *Query Extension* est chargé de modifier la requête fournie en entrée en considérant

[1] Le prototype est disponible à l'adresse suivante : `http://projets.upf.pf:8080/ProxyServer`. Identifiant : guest, Mot de passe : GuestPass.

un par un les différents motifs de triplet qui la composent. Chaque motif de triplet est soumis au module *Policy Engine* qui doit évaluer la politique de sécurité en fonction du motif et d'informations contextuelles supplémentaires. Le module *Policy Engine* détermine alors les éléments à ajouter à la requête. Une fois l'ensemble des motifs de triplet examiné, la requête obtenue est soumise au *RDF Repository* et la réponse est retournée à l'utilisateur.

Cette implémentation possède plusieurs défauts. Rien ne permet de dire si la ré-écriture de requête est toujours possible (*completness*) ? Comment savoir si la ré-écriture de requête est correcte (*correctness*) ? Une fois la requête ré-écrite, la réponse contient-elle l'ensemble des triplets autorisés ? De plus, cette implémentation nécessite la présence d'un module de confiance côté client ce qui est difficile à mettre en place. Enfin, il n'est pas prévu dans cette implémentation de module d'administration de la politique de sécurité.

Dans [HH04], les auteurs présentent *The Personal Reader Framework* (voir Fig1). Il s'agit d'un intermédiaire entre différents services de syndication (*Syndication services*) et des services de personnalisation (*Personalization services*). Ces deux types de services reposent sur des documents RDF. L'élément clé de ce *framework* est le *Connector Service* qui fait le médiateur entre deux instances de ces services. Le service de syndication, via le *Connector Service*, accède aux informations de certains domaines de façon personnalisée. Pour savoir comment personnaliser les informations, le *Connector Service* accède à un *User Modeling Service* qui contient les profils des utilisateurs (documents RDF). Ces informations sont transmises aux deux premiers services qui les utilisent pour plus de pertinence dans la sélection des informations.

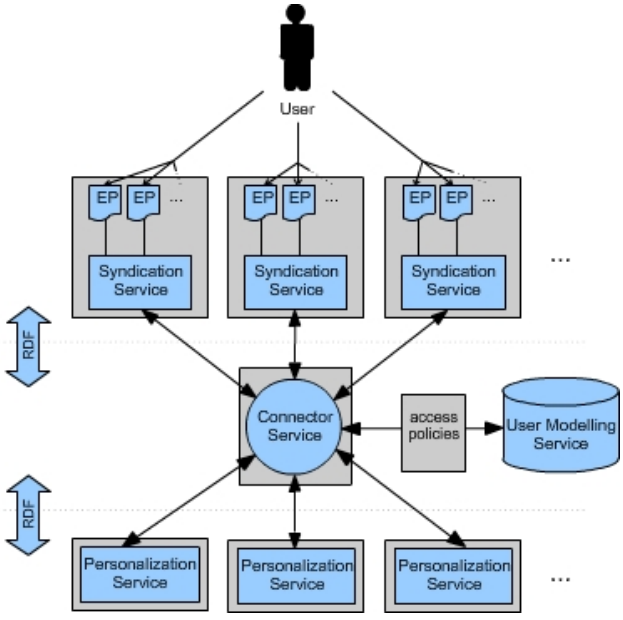

Fig. 1: Schéma du *Personal Reader Framework*

Dans [ACH+09], les auteurs reprennent la précédente architecture en ajoutant entre le *Connector Service* et l'*User Modeling Services* un contrôle d'accès. Ils présentent une interface utilisateur permettant de définir les règles de sécurité. Les utilisateurs disposent d'une interface via laquelle ils énoncent les règles qui protégeront leur profil. Les auteurs utilisent le langage Protune pour exprimer le règlement de sécurité.

The Personal Reader Framework ne proposait pas de contrôle d'accès avant l'ajout présenté dans [ACH+09] qui a l'avantage de disposer d'une interface d'administration. Cependant, une fois encore celle ci ne permet de définir que des règles de contrôle d'accès basées sur des motifs de triplet. L'énoncé des règles de sécurité se fait à l'aide de règles *deny* et *allow*. La résolution des conflits est dictée par la logique *deny-takes-precedence*. Cette implémentation prend en compte certaines informations contextuelles lors de l'évaluation du contrôle d'accès. Encore une fois, la politique de sécurité doit être évaluée pour chaque motif de triplet contenu dans une requête.

Notre implémentation ne souffre pas des mêmes lacunes que celles présentées ici. Contrairement à [ALC+07] qui modifient les requêtes, nous avons préféré les évaluer sur des sous-graphes du document RDF d'origine. La définition de ces sous-graphes se fait simplement et à l'aide d'un unique énoncé, sous forme d'une requête SPARQL (cf section 3). Contrairement aux précédents modèles où les motifs de triplet sont sélectionnés un par un, nous arrivons au même résultat en un seul énoncé. D'autre part, notre solution permet d'utiliser toute la sémantique des documents RDF pour construire ce sous-graphe.

Dans notre implémentation, la politique de sécurité n'est évaluée qu'une seule fois par requête et cette dernière peut être rejetée avant d'être évaluée. Soit une requête est rejetée directement, soit elle est évaluée sur un document où les triplets sont tous accessibles. Il n'y a ni modification de la requête, ni modification du résultat (cf. [Red05]).

D'un point de vue technique, la majeure partie des implémentations proposées utilisent JENA[jen], ou SESAME[ses] comme moteur de requête et/ou comme *RDF store*. Nous avons utilisé SESAME dans notre application car il est disponible sous forme d'une application web et propose un *RDF store* et une interface utilisateur très utiles pour effectuer des tests.

3 Modèle de sécurité

3.1 Modèles existants

Notre modèle de contrôle d'accès emprunte des caractéristiques dans de nombreux modèles existants. Nous rappelons brièvement ces modèles dans les sous-sections suivantes avant de décrire notre modèle de contrôle d'accès.

3.1.1 VBAC

SQL (Structured Query Language) est un langage permettant d'interroger et d'administrer les bases de données. Il est possible avec SQL de définir des vues à partir des informations contenues dans une ou plusieurs tables. En pratique, une vue correspond à une requête dans la base de données. Le système de sécurité de SQL est à la fois basé sur un système de rôles (RBAC : *Role Based Access Control*[FK92] [SCFY96]) et un système de vues (VBAC : *View Based Access Control*). Ainsi il est possible d'attribuer des droits sur une vue à certains rôles. L'attribution de droits se fait à l'aide de la commande GRANT.

Notre modèle adopte le système de vues du modèle de sécurité SQL. Ainsi, nous utilisons des requêtes SPARQL de types CONSTRUCT et DESCRIBE pour définir les différentes vues nécessaires. Utiliser les vues permet de réduire considérablement le nombre de règles dans la politique de sécurité. Grâce aux vues, il est possible de sélectionner en une seule requête plusieurs motifs de triplet, mieux encore, nous pouvons utiliser les liens entre les triplets lors de la construction de ces requêtes. Les vues sont les objets à sécuriser. Pour définir notre réglement de sécurité nous reprenons les mécanismes d'attribution, révocation de droits à la GRANT/REVOKE.

3.1.2 DAC et RBAC

Le modèle de contrôle d'accès DAC (*Discretionary access control*)[TCSEC85] est basé sur la notion d'identité, nous retenons la notion de droit propriétaire qui permet d'attribuer un propriétaire aux objets.

RBAC est un modèle de contrôle qui ne se base plus directement sur les utilisateurs, mais sur les rôles qui leur sont attribués. On passe donc d'un grand nombre d'utilisateurs à seulement quelques rôles bien définis. Ces rôles sont ordonnés de façon hiérarchique. L'énoncé et l'administration des différentes règles de contrôle d'accès s'en trouvent simplifiés.

Notre modèle de contrôle d'accès utilise la notion de propriétaire et permet la création de rôles.

3.1.3 ABAC

Le modèle ABAC (*Attibute-Based Access Control*)[YT05] a été proposé pour répondre aux faiblesses des modèles de contrôle d'accès existants lorsque l'on se trouve dans un environnement décentralisé. En effet, les modèles précédents requièrent l'authentification, or celle ci n'est pas simple à mettre en œuvre ni obligatoire dans le contexte d'une application web. Les règles de contrôle d'accès peuvent dépendre de l'authentification d'un utilisateur (et des rôles qui lui sont attribués dans ce cas), mais aussi d'attributs environnementaux tels que l'adresse IP (qui peut définir sa position géographique), l'heure de connexion. . .Le modèle ABAC, plus générique que DAC, permet de définir un utilisateur par un ensemble d'attributs.

3.2 Notre modèle de sécurité

Après avoir rappelé quelques principes dont nous nous inspirons, nous présentons dans cette section les détails de fonctionnement du modèle.

Pour bien définir notre modèle de sécurité, il faut tout d'abord définir les sujets, les objets et les actions. Les objets étant les éléments à protéger, et les sujets ceux qui exécutent les actions. Ici les sujets sont les utilisateurs. Les objets sont les documents RDF ou les vues définies sur ces documents. Enfin, les seules actions possibles sont les différents types de requêtes SPARQL : SELECT, ASK, CONSTRUCT, DESCRIBE.

3.2.1 Définition de la politique de sécurité

Dans cette section nous présentons le langage permettant d'énoncer les différentes règles de contrôle d'accès. Nous dérivons notre langage de celui utilisé dans le domaine des bases de données SQL. Dans notre modèle, l'énoncé d'une nouvelle règle de sécurité se fait à l'aide d'un PERMIT. Cependant là où SQL n'est basé que sur l'identité d'un utilisateur,

notre PERMIT autorise l'utilisation de conditions contextuelles (contexte temporel par exemple). La syntaxe d'un PERMIT dans notre langage est la suivante :

`C` → **PERMIT** (`U|PUBLIC`, `D|ALL`, `O`) **IDENTIFIED BY** `N`

Dans cette notation, `D` représente la liste des privilèges autorisés (types de requête SPARQL). Le mot clé `ALL` est équivalent à la liste des 4 types de requêtes. `O` est l'objet sur lequel les droits pourront être appliqués (une vue ou un document RDF). `U` est une variable qui doit apparaître dans `C`. Si `U` est remplacée par le mot `PUBLIC` alors la règle de sécurité s'adresse à tout le monde. `C` est une expression logique qui doit être vérifiée pour pouvoir bénéficier des droits `D` sur l'objet `O`. `C` est facultative si le mot clé `PUBLIC` est présent, sa présence est requise si le mot clé `U` est utilisé. Finalement, `N` est le nom donné à cette règle. Ce nom est utilisé lorsque l'on souhaite supprimer une règle.

La sémantique de cette règle est la suivante : l'utilisateur `U` a les droits `D` sur l'objet `O` tant que `C` est évalué vrai.

Pour retirer un droit, il suffit de supprimer la règle correspondante. Nous utilisons la syntaxe suivante :

DELETE `N` **FROM** `O`

La règle `N` est retirée de l'objet `O`.

Voyons maintenant quelques exemples de règles de contrôle d'accès. Supposons qu'Alice soit un utilisateur enregistré ayant mis à disposition sur le proxy un document RDF. Alice a aussi défini plusieurs vues et une hiérarchie de rôles.

`(TIME<20) AND (TIME>8)` → **PERMIT** (`PUBLIC`, `SELECT`, `minimalView`) **IDENTIFIED BY** `guestRule`
Dans cet énoncé, le mot clé `TIME` fait référence à l'heure courante (sur le système). Cette règle signifie donc qu'Alice autorise tout le monde (mot clé `PUBLIC`) à effectuer une requête de type `SELECT` sur sa vue `minimalView` durant la journée (entre 8 heures et 20 heures).

`PlayRole(X, FRIEND)` → **PERMIT** (`X`, `CONSTRUCT`, `FriendView`) **IDENTIFIED BY** `friendRule`
Ici la fonction `PlayRole` est utilisée pour vérifier que l'utilisateur dispose bien d'un certain rôle, ainsi `PlayRole(X, FRIEND)` est vrai si l'utilisateur (`X`) dispose du rôle `FRIEND` défini par Alice. Cette règle de sécurité autorise donc les utilisateurs qui sont amis d'Alice à effectuer des requêtes `CONSTRUCT` sur la vue `FriendView`.

`Identity(X, Bob)` → **PERMIT** (`X`, `ALL`, `minimalView`) **IDENTIFIED BY** `bobRule`
Dans cet exemple, la fonction `Identity` permet de vérifier l'identité de l'utilisateur. `Identity(X, Bob)` ne sera vrai que si l'utilisaeur s'est authentifié sur le proxy et qu'il s'agit bien de Bob. Cette règle de sécurité autorise donc l'unique utilisateur Bob à effectuer n'importe quel type de requête (`ALL`) sur la vue `minimalView`.

DELETE `friendRule` **FROM** `FriendView`
Enfin, voilà un exemple d'énoncé permettant de supprimer une règle présente dans la politique de sécurité.

Un certain nombre de modèles de sécurité prennent en compte des autorisations négatives ([ALC$^+$07] présenté dans la section 2 par exemple). Les autorisations négatives sont utilisées en règle générale afin d'exprimer une exception, une interdiction particulière à l'interieur d'une autorisation générale (ou inversement). Le principal défaut des autorisations négatives est qu'elles engendrent très facilement des conflits qui nécessitent d'être résolus avant de donner une réponse favorable ou non à la demande d'accès. La gestion des conflits est un problème difficile qui requiert souvent d'assigner un niveau de priorité aux règles. Dans notre cas, nous n'avons pas besoin de considérer les autorisations négatives. Lors de l'énoncé d'une nouvelle règle de sécurité, il suffit d'exprimer l'exception dans l'énoncé des conditions à valider. Maintenant que la règle `friendRule` a été supprimée, Alice souhaite la ré-écrire pour autoriser tous ses amis sauf Bob à accéder à cette vue avec le droit `CONSTRUCT`. La règle devient donc :

`PlayRole(X, FRIEND) AND NOT Identity(X, Bob)` →
PERMIT (`X`, `CONSTRUCT`, `FriendView`) **IDENTIFIED BY** `friendRuleBis`

3.2.2 Administration

Dans notre modèle de sécurité, chaque utilisateur est propriétaire des documents qu'il publie. Il a donc à charge de définir un ensemble de vues (à l'aide de requêtes CONSTRUCT et/ou DESCRIBE) et une hiérarchie de rôles.

Les différentes tâches d'administration consistent en la gestion des rôles et la gestion des différentes règles de sécurité.

Nous utilisons une syntaxe identique à celle de SQL pour la création de nouveaux rôles. Ainsi la commande :
CREATE ROLE `R`
est utilisée pour créer un nouveau rôle.

Seul l'administrateur peut attribuer et retirer les rôles qu'il a définis, ces rôles sont transmis soit à un autre rôle, soit à un utilisateur directement. Nous utilisons les deux énoncés suivants pour respectivement attribuer et retirer un rôle à un utilisateur ou à un autre rôle :
GRANT `role` **TO** `user|role`
REVOKE `role` **FROM** `user|role`
Par exemple, avec l'énoncé suivant, Bob se voit attribuer le rôle `FRIEND`.
GRANT `FRIEND` **TO** `Bob`

Il faut noter que dans notre modèle, les rôles sont gérés comme de simples attributs. Les droits ne sont pas attribués à certains rôles, mais aux utilisateurs qui disposent de ces rôles (cf règle `FriendRule` du paragraphe précédent).

Notons également que les rôles sont *locaux* à l'utilisateur qui les a définis. Plusieurs utilisateurs peuvent donc parfaitement définir un même rôle `Friend` sans problème de conflit. Les rôles créés par un utilisateur n'ont de valeur que dans le schéma de cet utilisateur. Lorsqu'un utilisateur crée une règle de contrôle d'accès, il ne peut utiliser que les rôles qu'il a lui même définis.

Une fois les rôles et vues créés, le propriétaire peut alors créer les règles de sécurité à l'aide des commandes définies dans la section précédente.

Le modèle de sécurité SQL propose une clause optionelle, WITH GRANT OPTION, permettant le transfert de droits. Dans notre modèle nous n'avons pas une telle clause. Il

est cependant tout à fait possible de déléguer des droits. Pour déléguer des droits sur une certaine vue, il suffit d'autoriser les requêtes de type CONSTRUCT sur cette vue. Ainsi un utilisateur disposant du droit CONSTRUCT pourra créer ses propres vues sur cette vue. Il sera alors propriétaire de ces vues qu'il pourra administrer. Il disposera de tous les privilèges et pourra définir ses propres rôles et vues.

4 Implémentation

Pour vérifier la faisabilité de notre modèle, nous l'avons implementé et testé. Dans cette section, nous présentons les choix d'architecture et d'outils que nous avons faits.

4.1 Architecture

Dans notre implémentation, nous utilisons des ACL (Access Control List) pour implémenter les règles. Concrètement, à chaque vue et à chaque document RDF est attaché un fichier ACL contenant l'ensemble des règles de contrôle d'accès le concernant. Le tableau en Fig2 représente par exemple le fichier ACL de la vue `FriendView` après l'exécution des différentes règles présentées dans la section précédente.

DROIT	**CONDITION**	**NOM**
OWNER	Identity(X, Alice)	OwnerRule
CONSTRUCT	PlayRole(X, Friend) AND NOT Identity(X, Bob)	FriendRuleBis

Fig. 2: Fichier ACL de la vue `FriendView`

La première règle est générée automatiquement lors de la création de la vue. La règle `ruleOwner` stipule qu'Alice est le propriétaire de cet objet, par conséquent, elle dispose de tous les droits sur celui-ci.

Nous avons choisi d'implémenter notre modèle de sécurité sous la forme d'un proxy. Un schéma de l'architecture générale est présenté sur la Fig3.

Notre proxy dispose de trois bases de données principales.

La première (*proxy Database*) est utilisée pour l'administration du proxy. Elle contient les informations sur les différents comptes utilisateurs enregistrés sur le proxy. La création d'un compte et l'authentification sont facultatives pour la plupart des utilisateurs. Cependant elle est indispensable pour les utilisateurs souhaitant publier des données RDF. Cette première base de données contient donc l'identifiant et le mot de passe des utilisateurs (et d'autres informations plus administratives telles que l'IP, l'adresse mail...).

La seconde base de données (*security policy database*) contient les hiérarchies de rôles, un annuaire des documents RDF et des vues ainsi que les différents fichiers ACL. Chaque utilisateur qui publie des données RDF dispose d'un schéma dans cette base de données. Ce schéma contient la définition de ses rôles et de ses objets.

La dernière base de données contient l'ensemble des documents RDF stockés de façon native.

Enfin, chaque utilisateur enregistré dispose d'un dossier qui contient les requêtes SPARQL de type CONSTRUCT ou DESCRIBE qu'il a créées sous forme de fichiers texte.

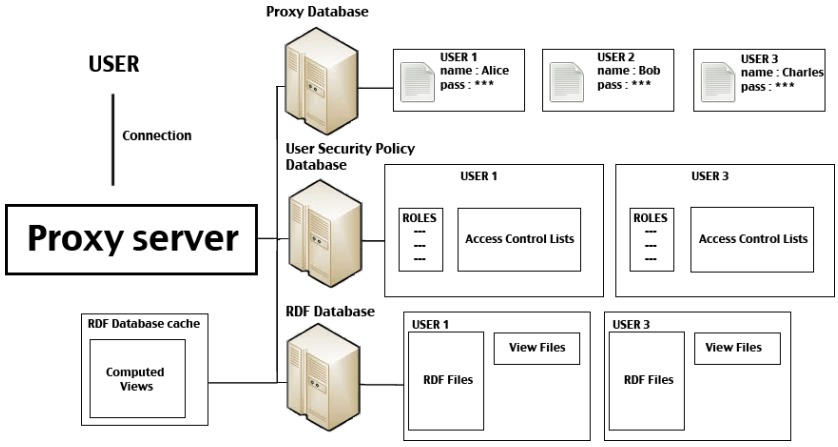

Fig. 3: Schéma du proxy

Dans notre implémentation, les vues sont évaluées dynamiquement et sont supprimées une fois la requête effectuée.

Chaque construction de vue nécessite d'effectuer une requête dans un document RDF. Ainsi le temps nécessaire à l'évaluation d'une requête dépend du nombre de vues intermédiaires qu'il aura été nécessaire d'évaluer pour construire le graphe RDF correspondant à la vue sur laquelle la requête utilisateur est effectuée.

Pour optimiser notre serveur, notre prototype intègre un cache mémoire. Ce cache est utilisé afin de garder en mémoire le résultat de l'évaluation des vues les plus solicitées. Ce cache améliore l'efficacité de notre implémentation puisqu'il n'est plus nécessaire d'évaluer systématiquement les vues les plus populaires. Seules les vues peu interrogées sont évaluées. Notre cache fonctionne très simplement à l'aide d'un compteur de popularité. À chaque requête, le compteur de toutes les vues qu'il aura été nécessaire d'évaluer est incrémenté. Le cache ne conserve que les vues dont le compteur est le plus élevé.

4.1.1 Comparaison avec l'architecture PEP/PDP

La plupart des implémentations de mécanismes de contrôle d'accès repose sur le paradigme **PEP/PDP**[LPA02] (*Policy Enforcement Point, Policy Decision Point*). Le PDP intercepte la demande d'accès et la transmet au PDP. Le PDP analyse la demande d'accès et détermine à l'aide des informations dont il dispose (contextes utilisateur et environnementaux) si l'accès est autorisé ou non. En fonction de la reponse du PDP, le PEP évalue ou non la requête initiale.

Notre prototype reprend les mêmes mécanismes de contrôle d'accès. Voici l'algorithme que nous utilisons :
L'utilisateur `u` soumet la requête `q`. Nous utilisons les notations suivantes :

- `s(q)` retourne la vue ou le document RDF interrogé par la requête `q`.
- `q(v)` retourne la requête CONSTRUCT ou DESCRIBE défini par la vue `v`.
- `o(v)` retourne le propriétaire de la vue `v`.
- `t(v)` retourne le type (SELECT, CONSTRUCT, ASK, DESCRIBE) de la requête

q.

```
If decision(u,q) then
   output(enforce(q))
Else
   output("Access Denied")
```

Notre PDP évalue la fonction (récursive) `decision`. Elle prend en paramètre un utilisateur et une requête, elle retourne un booléen.

```
Function decision(u,q) : Boolean
v ← s(q)
If v is an RDF graphe and Permit(u, t(q), v) can be derived from the
Security Policy then
   return True
Else /* v is a view */
   If Permit(u, t(q), v) can be derived from the Security Policy then
      q ← q(v)
      u ← o(v)
      return decision(u,q)
   Else
      return False
```

Dans cette fonction, la variable `u` n'est pas nécessairement instanciée. Ce n'est le cas que si l'utilisateur s'est authentifié sur le serveur. Notons aussi qu'il ne suffit pas q'un utilisateur ait les droits suffisants sur la vue qu'il interroge pour obtenir un resultat. Si le le propriétaire de la vue interrogée n'a pas les droits suffisants pour évaluer dynamiquement sa vue, alors, l'accès sera refusé (les permissions étant dépendantes des contextes, elles ne sont pas toujours activées).

Notre PEP utilise l'algorithme suivant. Il prend en entrée une requête et retourne le résultat de l'évaluation de cette requête (un document SRD ou un document RDF).

```
Function enforce(q) : RDF Graph or SRD
v ← s(q)
If v is an RDF graphe then
   return result of query q on v
Else /* v is a view */
   return result of query q on enforce(q(v))
```

Notre modèle dispose aussi d'un PAP (*Policy Administration Point*) qui permet, via une interface, à l'utilisateur de modifier la politique de sécurité contrôlant l'accès à ses documents.

4.2 Outils

Nous avons choisi d'implémenter notre modèle de sécurité comme une application web. Il s'agit là de la façon la plus simple de disposer d'une interface ouverte et accessible à tout ceux qui souhaitent partager leurs documents.

Notre application web est donc déployée dans un serveur d'applications Tomcat[tom], ce dernier repose sur le serveur web Apache[apa].

Nous utilisons le *RDF store* fourni par SESAME [ses]. Le *RDF store* SESAME a l'avantage d'être lui même distribué comme une application web déployée dans Tomcat. Il dispose également d'une API Java très complète qui nous permet d'interagir avec les documents RDF stockés dans le *store*. Il est possible d'interroger les documents RDF via des requêtes SPARQL.

D'autre part, dans un souci de simplification et pour ne pas multiplier les formats de bases de données, tous nos fichiers sont au format RDF, l'alternative eut été d'installer un SGBD en plus du *RDF store* déjà présent. Nous présentons dans le schéma en Fig4 la structure du document RDF issue de la première base de données *proxy Database*. Cette base de données n'est interrogée que lorsqu'un utilisateur tente de se connecter. Une simple requête SPARQL permet de savoir si l'identifiant et le mot de passe fournis correspondent bien à un compte existant (exemple Fig5 pour l'authentification de l'utilisateur Alice). Si c'est le cas, les deux triplets RDF correspondant à l'adresse IP de l'utilisateur et à la dernière date de connexion sont mis à jour et l'identifiant de l'utilisateur est stocké dans une session Tomcat afin d'être ré-utilisé tout au long de sa navigation dans le proxy.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:admin="http://home.org/admin#">
  <admin:User>
    <admin:Name>Alice</admin:Name>
    <admin:SHAPass>aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d</admin:SHAPass>
    <admin:Mail>alice@mail.fr</admin:Mail>
    <admin:IP>127.0.0.1</admin:IP>
    <admin:CreateDate>Thu Jan 21 16:50:04 TAHT 2010</admin:CreateDate>
    <admin:LastLogin>Thu Jan 21 16:50:04 TAHT 2010</admin:LastLogin>
  </admin:User>
  <admin:User>
    <admin:Name>Bob</admin:Name>
    <admin:SHAPass>7c211433f02071597741e6ff5a8ea34789abbf43</admin:SHAPass>
    <admin:Mail>bob@mail.pf</admin:Mail>
    <admin:IP>10.1.9.57</admin:IP>
    <admin:CreateDate>Thu Jan 21 16:52:27 TAHT 2010</admin:CreateDate>
    <admin:LastLogin>Thu Jan 21 16:52:27 TAHT 2010</admin:LastLogin>
  </admin:User>
</rdf:RDF>
```

Fig. 4: *Proxy Database*

```
1  PREFIX admin :<http ://home.org/admin#>
2  PREFIX rdf :<http ://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  ASK WHERE {
4   ?x rdf :type admin :User .
5   ?x admin :Name "Alice" .
6   ?x admin :SHAPass "SHA_PASS" .
7  }
```

Fig. 5: Validation de l'authentification

Les objets de la deuxième base de données sont également stockés sous la forme de documents RDF. La structure d'un exemple de document RDF contenant ces données est présentée dans la Fig6. L'ensemble des rôles, des documents RDF et des vues sont tout d'abord listés. Puis l'ensemble des règles de contrôle d'accès et d'attribution de rôle est énuméré. Lorsqu'un utilisateur effectue une requête sur une vue ou un document RDF, le PDP évalue si la requête est valide ou non en fonction des conditions définies dans le fichier ACL correspondant, stocké dans cette base de données.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#
         xmlns:acl="http://home.org/acl#">
ROLES
    <acl:Role rdf:nodeID="ro_Client">
            <acl:Name>Client</acl:Name>
    </acl:Role>
    <acl:Role rdf:nodeID="ro_Friend">
            <acl:Name>Friend</acl:Name>
    </acl:Role>

VIEWS
    <acl:RDFData rdf:nodeID="rd_myFoafFile">
            <acl:Name>myFoafFile</acl:Name>
            <acl:File>myFoafFile.txt</acl:File>
    </acl:RDFData>
    <acl:View rdf:nodeID="v_FriendView">
            <acl:Name>FriendView</acl:Name>
            <acl:File>FriendView.txt</acl:File>
            <acl:onData rdf:nodeID="rd_myFoafFile"/>
            <acl:ViewType>CONSTRUCT</acl:ViewType>
    </acl:View>
    <acl:View rdf:nodeID="v_minimalView">
            <acl:Name>minimalView</acl:Name>
            <acl:File>minimalView.txt</acl:File>
            <acl:onData rdf:nodeID="rd_myFoafFile"/>
            <acl:ViewType>CONSTRUCT</acl:ViewType>
    </acl:View>

    <acl:Rule>
            <acl:Type>GranRole</acl:Type>
            <acl:Role rdf:nodeID="ro_Friend"/>
            <acl:To>
                    <acl:User>
                            <acl:Name>Bob</acl:Name>
                    </acl:User>
            </acl:To>
    </acl:Rule>
ACL
    <acl:Rule>
            <acl:Type>Permit</acl:Type>
            <acl:Name>guestRule</acl:Name>
            <acl:Right>SELECT</acl:Right>
            <acl:On rdf:nodeID="v_minimalView"/>
            <acl:When>(Time&lt;8) AND (Time&gt;20)</acl:When>
    </acl:Rule>
    <acl:Rule>
            <acl:Type>Permit</acl:Type>
            <acl:Name>bobRule</acl:Name>
            <acl:Right>SELECT</acl:Right>
            <acl:Right>ASK</acl:Right>
            <acl:Right>DESCRIBE</acl:Right>
            <acl:Right>CONSTRUCT</acl:Right>
            <acl:On rdf:nodeID="v_minimalView"/>
            <acl:When>(Identity=Bob)</acl:When>
    </acl:Rule>
    <acl:Rule>
            <acl:Type>Permit</acl:Type>
            <acl:Name>friendRuleBis</acl:Name>
            <acl:Right>CONSTRUCT</acl:Right>
            <acl:On rdf:nodeID="v_FriendView"/>
            <acl:When>(Playrole=Friend) AND (NOT Identity=Bob)</acl:When>
    </acl:Rule>
</rdf:RDF>
```

Fig. 6: *User Security Database*

4.3 Exemple d'utilisation

Pour illustrer cette section, nous présentons ici un exemple classique d'un utilisateur souhaitant effectuer une requête sur l'une des vues définies par un second utilisateur.

Supposons que notre utilisateur ait besoin de s'authentifier, le rôle PUBLIC ne lui suffisant pas pour la requête qu'il souhaite effectuer. Il fournit donc son identifiant : Bob et son mot de passe. Une première requête est effectuée sur la première base de données *proxy Database*. La requête a été présentée plus tôt (Fig5).

Si la requête retourne vrai, alors Bob est authentifié et le proxy lui propose la page d'administration de ses documents RDF. Puisque Bob souhaite simplement effectuer une requête, il se dirige vers la page *requête* qui lui propose la liste des vues et des documents RDF de chaque utilisateur. Il choisit une vue ou un document RDF, saisit sa requête et la soumet au proxy. Le PEP récupère les attributs de l'utilisateur (les éventuels rôles que Bob possède) et les attributs environnementaux. À l'aide de ces informations, le PDP détermine la validité de la requête. Pour cela il recherche dans le fichier ACL correspondant les conditions à valider.

Chaque ensemble de conditions est évalué l'un après l'autre à l'aide des attributs fournis par l'utilisateur (IP, heure, identité, rôles...). Si aucun des ensembles de conditions n'est validé alors la requête échoue et l'utilisateur en est informé. Dans le cas inverse, le proxy vérifie si la vue séléctionnée est présente dans le cache, si ce n'est pas le cas, la vue est

evaluée et le nouveau document RDF est stocké. La requête est ensuite effectuée et le résultat retourné à l'utilisateur.

4.4 Performances

Nous présentons maintenant les résultats de l'évaluation de notre prototype. Pour nos tests, nous utilisons un document RDF contenant plus de 400000 triplets [INS].

Le graphe Fig7 présente le temps d'exécution d'une simple requête. Cette évaluation a été faite sans utilisation du cache mémoire. On peut voir que le temps d'exécution d'une requête dépend lineairement du nombre de vues qu'il aura été nécessaire d'évaluer avant de finalement pouvoir évaluer la requête de l'utilisateur. Lorsque le cache est présent, le temps d'exécution est linéairement dépendant du nombre de vues qui ne sont pas dans le cache.

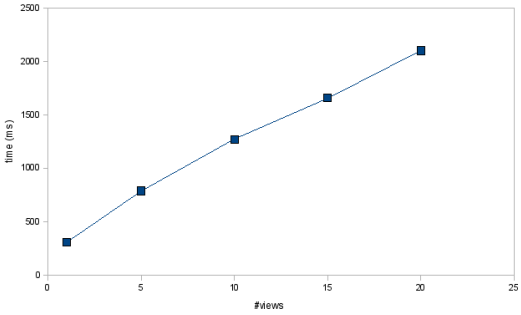

Fig. 7: Temps d'exécution d'une requête

Le graphe Fig8 présente maintenant le temps nécessaire à l'évaluation des droits. Une fois de plus, ce temps est linéaire par rapport au nombre de vues impliquées dans l'évaluation de la requête. Il apparaît que le temps nécessaire pour déterminer si une requête est autorisée ou non est négligeable par rapport au temps d'exécution d'une requête.

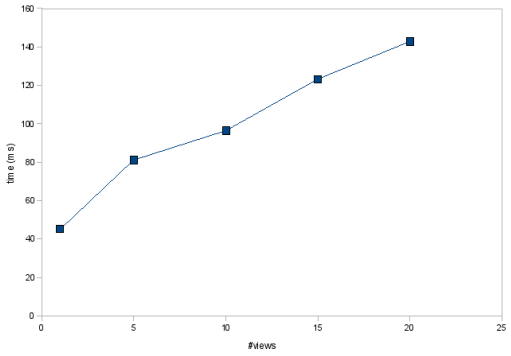

Fig. 8: Temps d'évaluation des droits

L'efficacité générale du modèle repose sur l'efficacité du moteur de requête SPARQL (SESAME dans notre cas).

5 Perspectives et conclusion

Dans cet article nous avons présenté notre prototype implémentant notre modèle de contrôle d'accès aux documents RDF. Il permet avec une interface utilisateur une administration décentralisée des documents RDF.

Outre l'amélioration de notre prototype, nous envisageons de suivre plusieurs pistes. Tout d'abord, une mise à jour du langage SPARQL fait l'objet d'un groupe de travail au sein du w3c[SM08]. La principale évolution sera la présence d'un nouveau type de requête : UPDATE. À l'aide d'une requête UPDATE, il sera possible de modifer un document RDF. Nous souhaitons naturellement ajouter ce type de requête dans notre protocole.

Nous envisageons également d'utiliser les attributs d'un utilisateur à l'interieur même de la définition des vues SPARQL. Par exemple en SQL il est possible de créer des vues contenant des variables système. Par exemple :

SELECT * **FROM** EMPLOYE **WHERE** nom=USER ;

Les variables système sont automatiquement instanciées par SQL. SPARQL ne permet pas une telle fonction. Nous souhaitons ajouter cette possibilité à notre prototype. Ainsi les attributs pourront être utilisés pour l'énoncé des vues. Les vues seront alors évaluées dynamiquement en fonction des utilisateurs.

Remerciements

Pour ces travaux, Léo Letouzey et Alban Gabillon sont financés par le projet ANR-SESUR-2007-FLUOR[flu].

Références

[AB04] Ben Adida and Mark Birbeck. Rdfa primer. Technical report, 2004.

[ACH+09] Fabian Abel, Juri Luca De Coi, Nicola Henze, Arne Wolf Koesling, Daniel Krause, and Daniel Olmedilla. A user interface to define and adjust policies for dynamic user models. In *WEBIST*, pages 184–191, 2009.

[ALC+07] Fabian Abel, Juri Luca, De Coi, Nicola Henze, Arne Wolf Koesling, Daniel Krause, and Daniel Olmedilla. Enabling advanced and context-dependent access control in rdf stores. In *International Semantic Web Conference 2007 (ISWC 2007), Busan, Korea*, 2007.

[apa] Apache software foundation home page. `http://www.apache.org/`.

[BB08] Dave Beckett and Jeen Broekstra. Sparql query results xml format. Technical report, 2008.

[BK03] Jeen Broekstra and Arjohn Kampman. Serql : A second generation rdf query language. In *In Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, pages 13–14, 2003.

[DA06] Sebastian Dietzold and Sören Auer. Access control on rdf triple stores from a semantic wiki perspective. In *In : Scripting for the Semantic Web Workshop at 3rd European Semantic Web Conference (ESWC*, 2006.

[FK92] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[flu] Fluor : convergence du contrôle de flux et d'usage dans les organisations. `http://fluor.no-ip.fr/index.php`.

[GL10] Alban Gabillon and Léo Letouzey. A view-based access control model for sparql. Soumis à NSS'10, 2010.

[HH04] Nicola Henze and Marc Herrlich. The personal reader : A framework for enabling personalization services on the semantic web. In *In Proceedings of the Twelfth GIWorkshop on Adaptation and User Modeling in Interactive Systems (ABIS 04)*, 2004.

[INS] INSEE. Code officiel géographique 2006 (cog-2006). `http://rdf.insee.fr/geo/`.

[Jai06] Amit Jain. Secure resource description framework : an access control model. In *In : ACM SACMAT*, pages 121–129, 2006.

[jen] Jena – a semantic web framework for java. `http://jena.sourceforge.net/`.

[KAC+02] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, Forth Vassilika Vouton, and Michel Scholl. Rql : A declarative query language for rdf. pages 592–603. ACM Press, 2002.

[LPA02] Hal Lockhart, Bill Parducci, and Anne Anderson. Oasis xacml tc, 2002.

[PS08] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf. Technical report, 2008.

[Red05] Pavan Reddivari. Policy based access control for a rdf store. In *In Proceedings of the Policy Management for the Web Workshop, A WWW 2005 Workshop*, pages 78–83, 2005.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2) :38–47, 1996.

[Sea04] Andy Seaborne. Rdql - a query language for rdf. Technical report, 2004.

[ses] Sesame, openrdf home page. `http://www.openrdf.org/`.

[SM08] Andy Seaborne and Geetha Manjunath. Sparql/update, a language for updating rdf graphs. Technical report, 2008.

[TCSEC85] United States Department of Defense Trusted Computer System Evaluation Criteria. Technical report, 1985.

[tom] Apache tomcat home page. `http://tomcat.apache.org/`.

[YT05] E. Yuan and J. Tong. Attributed based access control (abac) for web services. In *2005 IEEE International Conference on Web Services, 2005. ICWS 2005. Proceedings.*, 2005.